CPSC 213 – Assignment 1

Numbers and Memory

Due:	Wednesday, Sep 21, 2016 at 11:59pm No late assignments accepted.

Overview

One preliminary goal for this assignment is for you to familiarize yourself with the computing environment we will be using this term. You will learn about the UNIX command line, how to transfer files to the department UNIX servers, and how to submit assignments using the *handin* program. You will also setup an Eclipse or IntelliJ project for the *Simple Machine* processor simulator.

The assignment consists of five parts. The marks for each part and for each question within the parts are listed as a percentage as like this [xx%].

The following additional files are needed for the assignment and must be downloaded separately.

- The file *a1_code.zip* contains
  - Endianness.java used in Part 3.
- The file *sm-student-213.zip* contains
  - The SM213, *Simple Machine*, simulator used in Parts 3 and 4.

We strongly encourage you to do assignments with a partner (just one). See the *How to Hand In* Section for important details on how you do this.

What You Need to Do

Part 1: The UNIX Command Line

You can get a UNIX Command Line (i.e., a *shell*) in one of three ways.

1. You can login to one of the department server machines.
2. If you have a Mac (or run Linux) you can get a shell directly on your computer.
3. If you run Windows you can get a shell by installing *Cygwin*.

Pick your environment. Then do the following.

1. Create a file called `HelloWorld.java` and edit it so that it will print "`Hello World`" when it *runs*. You'll do this by creating a static method called *main* that prints that string.
2. Compile this class from the command line using the `javac` command.
3. Run this class from the command line using the `java` command.
4. Follow the instructions in the last section to use *handin* to submit what you've done so far. Then use *handin* again to verify that your code was submitted properly. The goal here is to workout the process of running *handin*. You'll run *handin* again later when you finish the rest of the assignment

Part 2: Install the Simple Machine

Down load *sm-student-213.zip* and unzip it. It contains two *jar* files and three *zip* files.

- `SimpleMachine213.jar` is the reference implementation of the simulator. The classes in this jar are obfuscated so that you can't decompile them back to useful source code.

  Save this file. It will come in handy later.

  To run the reference implementation you simply execute the jar file as described in *Companion Section B.4*. To do so from the command line, for example, you enter the following command:

  ```
  java -jar SimpleMachine213.jar
  ```

- `SimpleMachineStudent.jar` contains a non-obfuscated version of the simulator, but with only skeleton implementations of the `Memory` and `CPU` classes that you will be implementing. And so, this version of the simulator currently will not work. You will add your code to this version over the next couple of assignments.

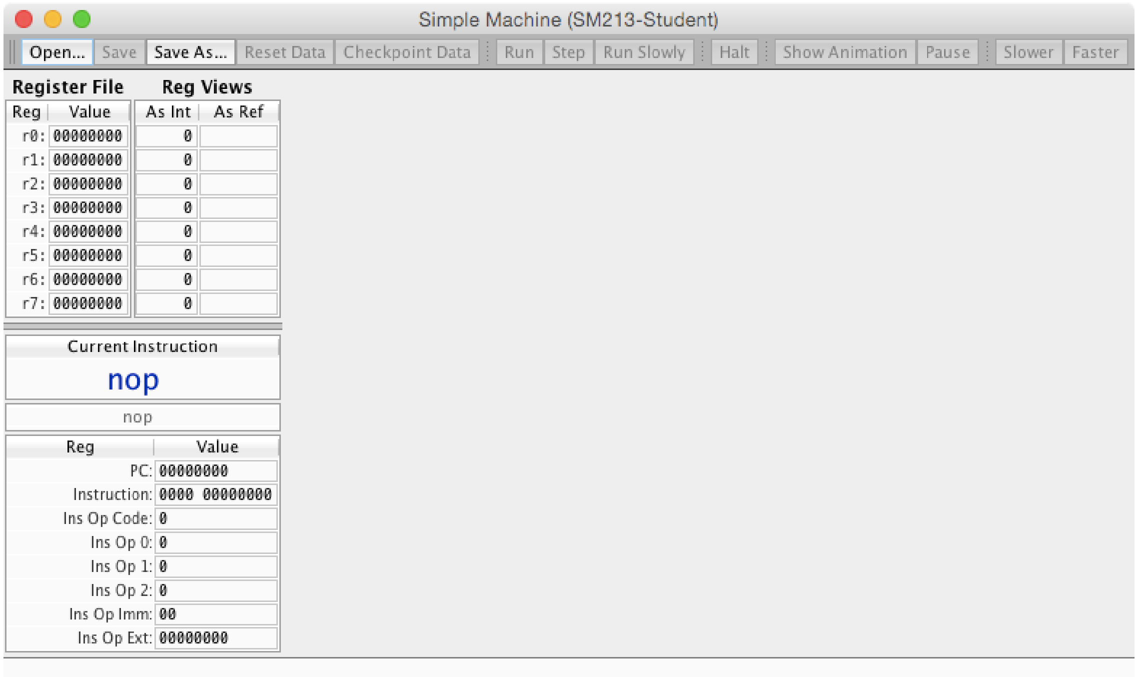
- `SimpleMachineStudentSrc.zip` contains the java source for the simulator including skeleton implementations of the two classes you will implement.

- `SimpleMachineStudentDoc213.zip` contains the *JavaDoc* for this source.

- `sm-student-213-eclipse.zip` is a pre-packaged Eclipse project for the simulator. This package was created for Eclipse Luna, Java 8. Hopefully this is the only file that you will need to use for this assignment.

Follow the instructions *Companion Section B.1* to install the Simulator into Eclipse. The other portions of the Appendix B give you additional instructions for creating your own Eclipse project from scratch using the other provided files. You should not *need* to do this; these extra files and those instructions are there just in case something goes wrong (e.g., you are using an older version of either Eclipse or Java).

Check to see whether your installation was successful by following the instructions in *Companion Section B.3* to run the simulator. If all went well, the simulator will start and show a



window like this:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
7000	73	C3	DA	3D	53	78	13	C7	F4	02	7D	2D	41	A0	C8	4B
7010	18	8D	25	06	9A	FC	35	70	87	B0	32	EF	41	1B	63	6C
7020	5D	29	FE	43	A7	B4	26	06	6D	A4	46	84	C8	1B	48	75
7030	57	C2	3E	78	C7	09	22	37	82	4B	0A	CC	37	53	7B	34
7040	64	07	A3	76	21	ED	DE	B2	21	B2	50	3A	CE	12	4D	E1
7050	52	45	AD	0A	FD	5C	8D	E7	15	EB	65	E6	05	6D	5E	02
7060	08	FD	01	2D	4D	70	A2	69	E1	66	13	56	B4	30	FF	A1
7070	04	21	C3	18	EE	B9	3C	5B	B0	37	0E	C2	CA	F8	4E	3A
7080	6C	53	AC	F3	4D	D6	2B	D2	57	C8	C6	C7	BD	5D	E2	FD
7090	EE	73	04	65	9B	8F	58	C0	A4	70	90	60	B1	2C	52	A6
70A0	A0	D7	1F	68	0F	0C	18	F4	8A	52	5F	1F	0E	1E	81	5E
70B0	7F	7E	4F	E3	35	F6	34	91	0A	D4	97	F7	FB	24	AE	0E
70C0	6E	5E	9E	20	19	75	B5	60	BC	66	74	91	BE	FF	7A	5A
70D0	7E	7C	E4	CE	37	43	0B	DA	6D	0D	45	9E	0C	C9	4A	6B
70E0	94	D9	4C	83	F5	82	66	7F	88	92	8A	9E	80	4A	8A	C9
70F0	37	45	BD	09	EA	B9	A9	75	29	80	5D	C3	EC	41	CD	62

Table 1: Contents of Memory from Address 0x7000 to 0x70FF.

Part 3: Endianness and Hex Questions [20%]

Answer these questions.

1. [4%] What is the value (in hex) of the little-endian, 4-byte integer stored at location `0x70d8` in Table 1?
2. [4%] What is the value (in hex) of the big-endian, 8-byte long stored at location `0x7020` in the table above?
3. [4%] Give an example of a 4-byte integer whose big- and little-endian representations are identical. Can you generalize this example?
4. [8%] The Hubble Space Telescope labels the image data it collects with sky positions using *right ascension* / *declination* coordinates. It downloads this data as binary files that are accessible on the Internet. You've decided that you'd like to take an up-close look at *Proxima Centauri*, the nearest star to earth after the sun, whose position is RA 14h 29m 42.9s, D -62 40 46.1

   Hubble image files encode position coordinates using two 4-byte integers, one for right ascension and the other for declination. And so the position of *Proximate Centauri* would be labeled as RA=521,829 and D=-2,207,359.

   ```
   RA =  14 * 36000 + 29 * 600 + 42.9 * 10 =    521,829
   D  = -62 * 36000 + 40 * 600 + 46.1 * 10 = -2,207,359
   ```

   So you write a program to download a portion of the Hubble dataset and search it for images containing these coordinates. You discover, however, that Hubble apparently never took any images of *Proximate Centauri*. You call the head of NASA to complain bitterly. She tells you that they have taken thousands of pictures of *Proxima Centauri* and suggest that perhaps you are an idiot.

   Then you note that the computer on the Hubble that generated the coordinates is the DF-224 manufactured by Rockwell Autonetics in the 1980's and the computer on which your program is running uses an Intel Core i7 processor that you recently purchased — and then you realize that something you learned in CPSC 213 might actually be useful.

   What did you realize and what are the correct values of the two integers that you should use in your program to search for *Proxima Centauri*?

   HINT: Use a calculator or a program to convert the numbers 521,829 and -2,207,359 to hex. Then think about how you might need to manipulate these hex values. You can give your answer in hex or convert it back to decimal; your choice. To print the hex value of a number in Java:

   ```
   System.out.printf("0x%x\n", i);
   ```

Part 4: Programming Test of Endianness [30%]

You are given the skeleton of an executable Java class in the file `Endianness.java`. Place this file directory on a UNIX machine (e.g., one of the lab machines, a Mac, or Windows running Cygwin) and compile it from the UNIX command line like this:

```
javac Endiannness.java
```

You can now run this program from the command line. The program takes four command-line arguments. These arguments are the values of four consecutive bytes (in hex) of memory from which the program will construct both big-endian and the little-endian integers. For example, to see the value of the integer whose byte values are 0x01, 0x02, 0x03, and 0x04, in that order, you would type:

```
java Endiannness 1 2 3 4
```

[25%] Write the code that transforms this memory into integers by replacing the `TODO`s with an implementation of `bigEndianValue` and `littleEndianValue.`

[5%] Test your program carefully by calling it from the command line with various memory values. Ensure that your tests provide good coverage and be sure to include some tests with bytes that have bit-eight set to 1 (e.g., 0xff or 0x80). Your mark here will be based the list of tests you ran, as described in the provided `P4.txt` file.

*Part 5: Implement the Simple Machine **MainMemory** Class* [50%]

Like a real processor, the simulator has a memory and a CPU. You will implement both of them as java classes. This week you will implement the memory.

Some portions of the memory are already implemented. Your job is to implement and test the five methods of `MainMemory.java` labeled with `TODO`'s. You will find this file in your Eclipse environment in the `arch.sm213.machine.student` package.

[35%] Implement the following methods.

- [10%] `isAligned` that determines whether an address is aligned.
- [10%] `bytestoInteger` and `integerToBytes` that translate between an array of bytes and a *big endian* integer. You can use your solution to Part 2 for this.
- [15%] `get` and `set` that provide array-of-byte access to memory used by the CPU to fetch and to store data. Follow the specification listed in the javadoc comments carefully. Note, for example, that the address provided to these methods is not required to be aligned.

[15%] Create a set of `JUnit` tests to test your implementation. Place all of your tests in a class named `MainMemoryTest` in the same package as the `MainMemory` class. Note that you will not actually be able to run the simulator itself yet beyond the initial screen, because you will still lack a CPU implementation.

Ensure that your tests provide good test coverage for each of five methods you implemented. Comment each test to explain what it is testing.

Summary of What to Hand In

Create a directory named `a1` that contains the following files:

1. `README.txt` – that contains the name and student number of you and your partner
2. `PARTNER.txt` – containing your partner's CS login id and nothing else (i.e., the 4- or 5- digit id in the form a0z1). Your partner should not submit anything.
3. `HelloWorld.java`
4. `Q1.txt` … `Q4.txt` – containing your answers to Questions 1-4, respectively.
5. `Endianness.java` – your solution to Part 4.
6. `P4.txt` containing your test description for Part 4.
7. `MainMemory.java` – your `MainMemory` solution
8. `MainMemoryTest.java` – your `JUnit` tests for Part 5.

Be sure this directory contains these files and nothing else and then follow the instructions in the next section to submit your assignment.

How to Hand In

You must use a program called `handin` to submit your assignment. To use this program you need a CS login id. If you don't have one you must get one. Your partner needs one too. To get a CS id following the online instructions here:

www.cs.ubc.ca/students/undergrad/services/account.

You should run `handin` program from the UNIX command line. Instructions for using the command-line version are here:

my.cs.ubc.ca/docs/handin-instructions

As indicated in the instructions, place your assignment files in a directory called `a1`.

You can hand in as many times as you want up to the grace-period deadline. The last hand in wins.

The instructions also tell you how to very that your hand in was successful. You should always double check.

The hand-in script will perform certain automated marking tasks and assign a tentative mark for certain parts of the assignment. But, since automated marking has limitations, TAs will review these automated marks and make corrections where appropriate.